MONTGOMERY MODULAR MULTIPLICATION ALGORITHM ON MULTI-CORE SYSTEMS

Junfeng Fan, Kazuo Sakiyama, and Ingrid Verbauwhede

Katholieke Universiteit Leuven,ESAT/SCD-COSIC,
Kasteelpark Arenberg 10
B-3001 Leuven-Heverlee, Belgium

ABSTRACT

In this paper, we investigate the efficient software implementations of the Montgomery modular multiplication algorithm on a multi-core system. A HW/SW co-design technique is used to find the efficient system architecture and the instruction scheduling method. We first implement the Montgomery modular multiplication on a multi-core system with general purpose cores. We then speed up it by adopting the Multiply-Accumulate (MAC) operation in each core. As a result, the performance can be improved by a factor of 1.53 and 2.15 when 256-bit and 1024-bit Montgomery modular multiplication being performed, respectively.

Index Terms— Montgomery Modular Multiplication, Multi-core, Parallel Architectures

1. INTRODUCTION

Modular multiplication is a fundamental operation in many popular Public Key Cryptography (PKC) algorithms such as RSA [1] and ECC [2, 3]. As the division operation in modular reduction is time-consuming, Montgomery [4] proposed a new algorithm where division is avoided. An integer X is represented as $X \cdot R \bmod M$, where M is the modulo and $R = 2^r$ is a radix which is coprime to M. This representation is called Montgomery residue. Multiplication is performed in this residue, and division by M is replaced with division by R.

So far, the Montgomery modular multiplication algorithm has been widely implemented in both software [5, 6, 7] and hardware [9, 10, 11]. Compared to the software implementations, the hardware implementations are faster as a dedicated data-path is used. However, they are fixed in functions and are not able to respond to new algorithms. The software implementations are flexible and can be easily modified to perform new algorithms, while they are not fast enough in some real-time applications. Therefore, combining the advantages of both software implementations and hardware implementations is necessary.

In this paper, we investigate the implementation of the Montgomery modular multiplication on a multi-core coprocessor. Multi-core processors are chose as the platform because they have multiple data-paths, and are completely programmable. We use a Very Long Instruction Word (VLIW) processor as a prototype. The Montgomery modular multiplication is accelerated by performing parallel computation. The bottleneck of this implementation is analyzed. We optimize the platform by deploying multiply-accumulate instruction in each core.

The rest of the paper is organized as follows. Section 2 briefly reviews previous work on the Montgomery algorithm and its implementations. In section 3, we describe the architecture of our platforms. The instruction scheduling method is proposed in section 4. Section 5 proposes a modified platform to speed up the computation. Finally, we show the implementation results in section 6 and conclude the paper including future work in section 7.

2. PREVIOUS WORK

The Montgomery modular multiplication algorithm was designed to avoid division in modular multiplications. Given two n-bit inputs, X and Y, this algorithm gives $Z = X \cdot Y \cdot R^{-1} \bmod M$, where R equals to 2^n and M is the n-bit modulo. Algorithm 1 shows the Radix-2^w Montgomery modular multiplication algorithm in detail. A modified Montgomery multiplication algorithm was proposed to avoid the conditional final substraction by choosing a suitable R [12].

As shown in Algorithm 1, the operands X, Y and M are divided into w-bit words. In the beginning of each iteration, $X_0 \cdot Y_i$ is calculated to generate T. After the generation of T, the multiplication of $X \cdot Y_i$ and reduction of C are performed together by doing $Z = Z + X \cdot Y_i + M \cdot T$. After that, Z_0 always becomes 0. The division of Z by r is performed by shifting Z one word to the right. After s iterations and one conditional substraction, $Z = X \cdot Y \cdot R^{-1} \bmod M$ is obtained. As Algorithm 1 scans the operands X and M from Least Significant Bit (LSB) to Most Significant Bit (MSB) simultaneously, it is also called Finely Integrated Operand Scanning (FIOS).

In recent years, the Montgomery modular multiplication has been widely implemented in software and hardware. For example, In [14], it was implemented on an 8-bit microcon-

Algorithm 1 Radix-2^w Montgomery modular multiplication (FIOS) [13]

Input: integers $M = (M_{s-1}, ..., M_0)_r$, $X = (X_{s-1}, ..., X_0)_r$, $Y = (Y_{s-1}, ..., Y_0)_r$, where $0 \leq X, Y < M$, $r = 2^w$, $s = \lceil \frac{n}{w} \rceil$, $R = r^s$ with $gcd(M, r) = 1$ and $M' = -M^{-1} \bmod r$.

Output: $X \cdot Y \cdot R^{-1} \bmod M$

1: $Z = (Z_{s-1}, ..., Z_0)_r \leftarrow 0$
2: **for** $i = 0$ to $s - 1$ **do**
3: $\quad T \leftarrow (Z_0 + X_0 \cdot Y_i) \cdot M' \bmod r$
4: $\quad Z \leftarrow (Z + X \cdot Y_i + M \cdot T)/r$
5: **end for**
6: **if** $Z > M$ **then**
7: $\quad Z \leftarrow Z - M$
8: **end if**
9: return Z

troller. In [6] it was implemented on an high-end TI DSP (TMS320C6201) . Großschädl [7] showed that the software implementations on general purpose CPU can be sped up by extending the ISA. These software implementations are highly flexible, whereas the performance is limited. The hardware implementations of the Montgomery multiplication were also widely investigated. Researchers have deployed various architectures, such as bipartite multipliers [15] and systolic arrays [16, 17, 18] to achieve high thoughput. In order to obtain some flexibility, reconfigurable datapath [11] for the Montgomery modular multiplication was also explored. However, there is still a gap between the flexibility and performance. One way to bridge the gap is using parallel computation with programmable devices, e.g., dual-mac DSP [6].

In this paper, we investigate the implementation of the Montgomery modular multiplication on a multi-core system. A VLIW processor with general purpose cores is proposed. According to the implementation result, we optimize it by deploying the MAC instruction in each core. A new instruction scheduling method is also introduced to achieve high parallelism.

3. OUR DESIGN PLATFORM

In order to achieve an efficient and flexible implementation, the HW/SW co-design method is used. A quick and correct evaluation of cost and performance for various hardware configurations and software programs is needed during the design process. Thus, we use a simulation environment, called GEZEL [20], which allows us to estimate immediate system performance in a cycle-accurate manner before synthesizing the entire design. The GEZEL code can be automatically converted to VHDL code and then synthesized.

Our first design platform, referred as platform-I, is a VLIW processor with general purpose cores. As shown in Figure 1, this platform consists of a main controller, a data memory, an

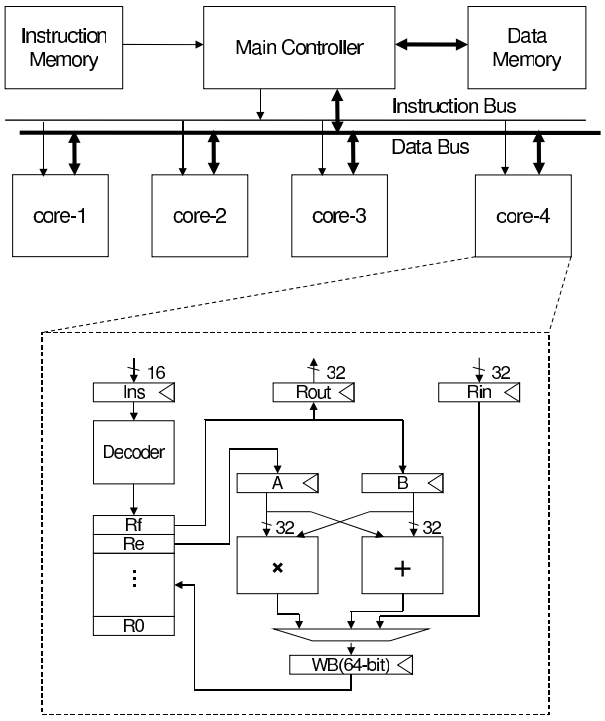

Fig. 1. Platform-I architecture. ($w = 32$).

instruction memory and several cores. Only the main controller can access the instruction memory and the data memory. The main controller fetches instructions from the instruction memory and dispatches them to all cores in parallel via the instruction bus. Each core executes arithmetic instructions in parallel, and stores the results in its register file. The data memory has only one read/write port, therefore, a single data memory access is allowed in each cycle.

The block diagram of the core is also shown in Figure 1. We denote w as the operation size of w-bit cores. It is a highly simplified Load/Store CPU. It has an instruction decoder, a register file with sixteen 32-bit registers and a status register. The Arithmetic Logic Unit(ALU) includes one 32-bit multiplier and one 32-bit adder. It also has an output register to store the data that will be written to the data memory, and an input register to buffer the data from the data memory. Both of them are 32-bit. One Write Back (WB) register is also used to store data from the ALU.

The cores here support a simple Load/Store Instruction Set Architecture (ISA). As shown in Table 1, this simplified ISA has only 8 general instructions. Here `#Addr` denotes a memory address. Instructions for each core are 16-bit long. All the arithmetic operations are performed among data stored in the register file. When data needs to be moved from one core to another, it is first stored to the data memory, then loaded by the destination core. Cores in this platform support a 4-stage instruction pipelining: namely, Instruction fetch and decoding, Register fetch, Execute and Register write back.

Table 1. Instruction sets for each core.

Opc 4-bit	*Opr* 1 4-bit	*Opr* 2 4-bit	*Opr* 3 4-bit	*Description*
`Nop`				No operation
`Load`	`Ri`	`#Addr`		Load the data from location `Addr` of the data memory into register `Ri`
`Store`	`Ri`	`#Addr`		Store the data of register `Ri` to location `Addr` of the data memory
`Mul`	`Ri`	`Rj`	`Rk`	`{R(i+1),Ri}=Rj· Rk`
`Add`	`Ri`	`Rj`	`Rk`	`{Ca,Ri}=Rj+Rk,Ca` is the carry out and is stored in the status register
`Adc`	`Ri`	`Rj`	`Rk`	`{Ca,Ri}=Rj+Rk+Ca`
`Sub`	`Ri`	`Rj`	`Rk`	`Ri=Rj−Rk−Ca`
`Suc`	`Ri`	`Rj`	`Rk`	Conditional `Sub`

4. INSTRUCTION SCHEDULING

The Montgomery modular multiplication algorithm is partitioned and mapped to each core. In order to achieve a high performance, the instructions are manually scheduled so that all the cores are utilized efficiently. The instruction scheduling method is the essential part of the software implementation.

The data dependency of the Montgomery algorithm is analyzed in Figure 2. The main dependency is due to the carries of additions. Taking FIOS shown in Algorithm 1 as an example, in each iteration, Z_j is replaced by $(Z_j + (X \cdot Y_i)_j + (M \cdot T)_j + Ca)$, where Ca is the carry. Obviously, $X_j \cdot Y_i$, for any $0 \leq i, j \leq s-1$, is only dependent on the operands X and Y. We can also calculate $M_j \cdot T$ immediately after the generation of T. The products with the same weight of Z_j and the carry from Z_{j-1} are accumulated to Z_j, generating a new Z_j and 2-bit carries. As a result, Z_j can only be generated after the carry from Z_{j-1} is ready.

As shown in Figure 2, we need to add Z_j with four w-bit data and 2-bit carries. In hardware implementations, cascaded Carry Save Adders (CSAs) can be used to construct a 6-to-2 CSA. The carry can also be saved in a 2-bit register or transferred to another PE. However, in general purpose processors these special features are not available. Normally only general adders with a fixed length are used. The carry is saved in the status register after an `Add` instruction. In order to keep the 1-bit carry for future use, one instruction is needed to copy it from the status register to a general register. It will be very inefficient to use carries generated by another core, since it needs to be stored to register file first, and then transferred via the data memory.

Therefore, it will be desirable to partition the algorithm so that carry is only used in the core where it was generated. Note

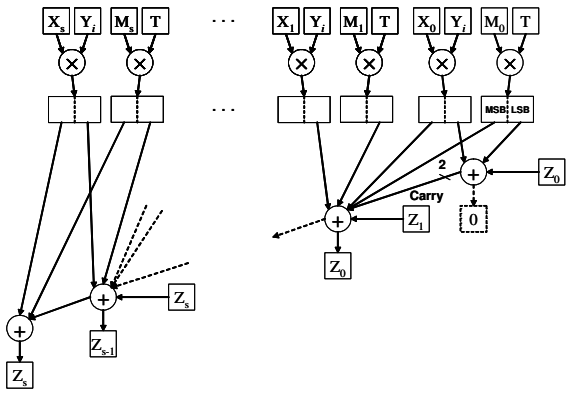

Fig. 2. Data dependency of FIOS Montgomery algorithm.

that in order to generate T, only Z_0 must be ready at the end of the previous iteration, while $(Z_{s-1}...Z_1)$ can be generated later. Based on this observation, an instruction scheduling method is proposed and is shown in Figure 3. In this method, each iteration in Algorithm 1 is performed by multiple cores. Here we choose $n = 256$, $w = 32$ and $s = \lceil \frac{n}{w} \rceil = 8$. During the whole loop (Z_1, Z_0) is generated and stored in core-1, (Z_3, Z_2) in core-2, (Z_5, Z_4) in core-3 and (Z_7, Z_6) in core-4. Carry is only used in the local core. At the end of each iteration, Z_1 is sent to core-1, Z_3 is sent to core-2 and Z_5 is sent to core-3. After 8 iterations and a conditional substraction, $Z = X \cdot Y \cdot R^{-1} \bmod M$ is generated and stored separately in four cores. Z can be written to the data memory or can be used by another modular multiplication.

This method has two advantages. First, it utilizes all the four ALUs efficiently by symmetrically partitioning the Montgomery modular multiplication algorithm. Second, operands and intermediate data are distributed in the register file of each core, thus less registers in each core are required. According to Figure 3, core-1 only needs to store (X_1, X_0), (M_1, M_0) and (Z_1, Z_0). During the whole computation they can stay in the register file. As a result, the number of `load` and `store` operation are reduced.

When using one core to perform 256-bit Montgomery modular multiplication, 644 clock cycles are required. When using 4 cores, we need only 217 clock cycles. That is, the 4-core based implementation is 2.96 times faster than the single-core based implementation.

The implementation result is summarized in Table 2. According to the table, the bottleneck of this implementation is addition operations. The number of addition operations is al-

Table 2. Number of each operation in one 256-bit Montgomery modular multiplication: Platform-I

`Mul`	`Add`	`Sub`	`Load`	`Store`	`Nop`	Total
136	372	16	113	44	187	217

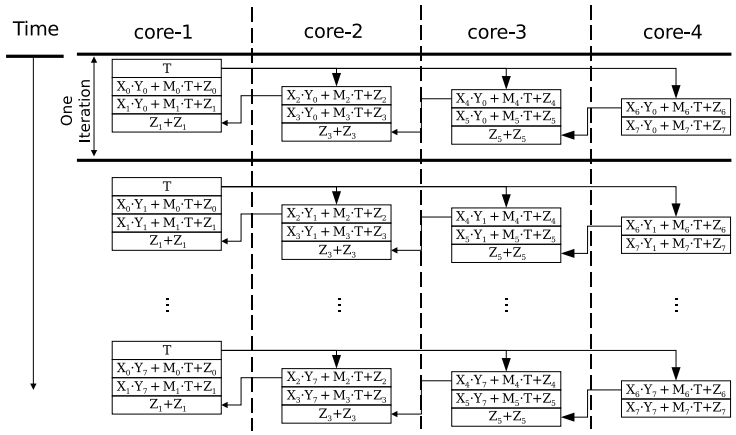

Fig. 3. Instruction scheduling method. ($n = 256$, $w = 32$, $s = \lceil \frac{n}{w} \rceil = 8$).

most three times larger than the number of multiplication. In the platform-I, one `Add` instruction consumes one clock cycle, just as one `Mul` instruction does. In order to improve the performance of this implementation, addition operations need to be accelerated.

5. PERFORMANCE SPEEDUP

As shown in section 1, in the k^{th} iteration we perform $(Ca, Z_{i+1}, Z_i) = X_i \cdot Y_k + M_i \cdot T + (Z_{i+1}, Z_i)$, where $0 \leq i, k < s$. This operation can be efficiently performed with two MAC operations, $(Ca, Z_{i+1}, Z_i) = X_i \cdot Y_k + (Z_{i+1}, Z_i)$ and $(Ca, Z_{i+1}, Z_i) = M_i \cdot T + (Z_{i+1}, Z_i)$. Here the Ca from the first MAC operation needs to be saved before being replaced by the second one. Based on this observation, we propose a revised multi-core platform, platform-II. Compared to the platform-I, cores in platform-II have one more 32-bit adder. The block diagram of the modified core is shown below.

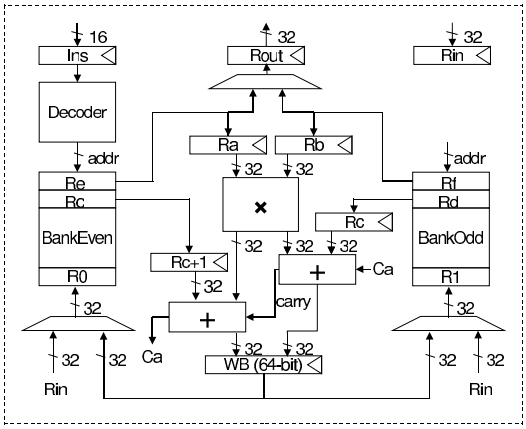

Fig. 4. Block diagram of the cores in platform-II. ($w = 32$).

In the platform-II, each core contains a multiplier and two adders. Besides the ISA shown in Table 1, two more instructions are supported by the platform-II.

```
MAC Rc,Ra,Rb
Adw Rc,Ra,Rb
```

Here we specify `Rc+1` implicitly. The `MAC` instruction performs `(Ca,Rc+1,Rc)=(Rc+1,Rc)+Ra*Rb+Ca`, and `Adw` instruction performs `(Ca,Rc+1,Rc)=(Rc+1,Rc)+(Ra,Rb)+Ca`. When executing `MAC` and `Adw`, we need to read four data from `Rc+1`, `Rc`, `Ra` and `Rb`, and write 2 data back to `Rc+1` and `Rc`. As a result, the register file needs four read ports and two write ports. As increasing the number of read/write ports causes drastic increment in area, register file with two separated banks, bank odd and bank even, are used. Each bank contains eight 32-bit registers and has one write port and two read ports. When performing `MAC` and `Adw` instructions, `Ra` and `Rb` are always in different banks, and so do `Rc+1` and `Rc`.

We use the same instruction scheduling method. The implementation result of the 256-bit modular multiplication is summarized in Table 3. The number of addition operation on the platform-II is only 30% of that on the platform-I. For one 256-bit modular multiplication, the number of cycles in total is about 42% less than that of the implementation on the platform-I.

Table 3. Number of each operation in one 256-bit Montgomery modular multiplication: Platform-II

Mac	Mul	Add	Sub	Load	Store	Nop	Total
128	8	114	16	93	32	109	125

6. RESULTS

The multi-core platform proposed in section 3 is implemented with GEZEL. The GEZEL code is automatically conver-

Table 4. Performance comparison of modular multiplication.

Reference	**Description**	**Platform**	**Area** (Slices)	**Freq.** (MHz)	**256-bit time**(μs)	**1024-bit time**(μs)
This work (Platform-I)	4-cores 4 32x32 mults	Xilinx XC2VP30	3873	93	2.3	44.0
This work (Platform-II)	4-cores 4 32x32 mults	Xilinx XC2VP30	4233	81	1.5	20.4
Tenca & Koç [5]	Software implementation	ARM processor	-	80	43	570
Cohen *et al.* [21]	Software implementation	UltraSPARC GMP library	-	143	14.6†	–
Itoh *et al.* [6]	Software implementation	DSP TMS320C6201	-	200	2.68‡	–
Brown *et al.* [22]	Software implementation	Pentium II	-	400	1.57§	–
Sakiyama *et al.* [10]	CSAs based Dual-Field	Xilinx XC2VP30	4836	110.4	0.80	–
Kelley *et al.* [9]	4-PEs 8 16x16 mults	Xilinx XC2V2000-6	360*	135	0.68	8.3
Mentens *et al.* [11]	34 16x16 mults	Xilinx XC2VP30	5500	125	0.17*	2.1

* Author's estimation from the original paper. † 224-bit modular multiplication.
‡ 239-bit Montgomery modular multiplication. § Using fixed modulo for fast reduction.

ted to synthesizable VHDL code. The software program of Montgomery modular multiplication is stored in the instruction memory. The operands, X, Y and M, are stored in the data memory.

For the purpose of checking the maximum frequency, the platform is implemented on Xilinx Virtex-II PRO (XC2VP30) FPGA. A maximum frequency of 93 MHz could be achieved for the platform-I and 81 MHz for the platform-II. The instruction memory and the data memory are implemented in the block RAM on the FPGA board. The number of slices here only includes the main controller and cores. The performance comparison between our software implementations and the state-of-the-art implementations is summarized in Table 4.

As shown in Table 4, the 256-bit modular multiplication on the platform-II is almost 28 times faster than the implementation on the ARM processor [5] and almost 9 times faster than the implementation on the UltraSPARC processor [21]. Compared to the implementation on TI's dual-mac DSP (TMS320C6201), our implementation is about 1.78 times faster. The implementation of [22] obtains a high performance, while only supports fixed modulo. Compared to the state-of-the-art hardware implementations [9, 10, 11], software implementations are still much slower. This is because of a dedicated datapath is used. For example, in [11] 34 multipliers are used and can finish one iteration of the Algorithm 1 in one clock cycle.

7. CONCLUSIONS

In this paper, we introduced an efficient software implementation of the Montgomery multiplication algorithm on a multi-core system. A prototype of general multi-core systems is implemented. We proposed a scheduling method and based on the implementation result a new platform is proposed to improve the performance. The new platform supports multiply-accumulate instructions and can accelerate the calculation by a factor of 1.53 and 2.15 when 256-bit and 1024-bit Montgomery modular multiplication are performed, respectively.

Our future work includes speeding up the data transfers between different cores and downsizing the whole platform. We believe that by improving the data transfer scheme a higher performance could be achieved without losing flexibility. This platform can also be used to perform other algorithms, e.g., modular inversion using Extended Euclidean Algorithm (EEA).

Acknowledgments

Junfeng Fan and Kazuo Sakiyama are funded by a research grant of the Katholieke Universiteit Leuven and FWO projects (G.0450.04, G.0475.05). This work was supported in part by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), by the EU IST FP6 projects (SESOC and ECRYPT), by the K. U. Leuven, and by the IBBT-QoE project

of the IBBT.

8. REFERENCES

[1] R. L. Rivest, A. Shamir and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120-126, 1978.

[2] N. Koblitz. Elliptic curve cryptosystem. *Math. Comp.*, 48:203-209, 1987.

[3] V. Miller. Uses of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology: Proceedings of CRYPTO'85*, number 218 in LNCS, pages 417-426. Springer-Verlag, 1985.

[4] P. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*,44:519-521,1985.

[5] A. Tenca and Ç. K. Koç. A scalable architecture for modular multiplication based on Montgomery's algorithm. *IEEE Transactions on Computers*, 52(9):1215-1221, September 2003.

[6] K. Itoh, M. Takenaka, N. Torii, S. Temma, and Y. Kurihara: Fast implementation of public-key cryptography on a DSP TMS320C6201. *Proceedings of Cryptographic Hardware and Embedded Systems - CHES'99*, LNCS 1717, pp. 61-72, Springer-Verlag, 1999.

[7] J. Großschädl, K. C. Posch, and S. Tillich. Architectural Enhancements to Support Digital Signal Processing and Public-Key Cryptography. *Proceedings of the 2nd Workshop on Intelligent Solutions in Embedded Systems (WISES 2004)*, pp. 129-143, Graz, Austria, June 25, 2004.

[8] S. E. Eldridge and C. D. Walter. Hardware implementation of Montgomery's modular multiplication algorithm. *IEEE Transactions on Computers*,42(6):693-699,June 1993.

[9] K. Kelley and D. Harris. Parallelized very high radix scalable Montgomery multipliers. *Conference on Signals, Systems and Computers*, pages 1196-1200, 2005.

[10] K. Sakiyama, B. Preneel and I. Verbauwhede. A fast dual-field modular arithmetic logic unit and its hardware implementation. *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS 2006)*, pages 787-790, 2006.

[11] N. Mentens, K. Sakiyama, B. Preneel, and I. Verbauwhede. Efficient Pipelining for Modular Multiplication Architectures in Prime Fields. *Proceedings of the 2007 Great Lakes Symposium on VLSI (GLSVLSI 2007)*, 2007.

[12] C. D. Walter. Montgomery's exponentiation needs no final subtraction. *Electronic letters*, 35(21):1831-1832, October 1999.

[13] Ç. K. Koç, T. Acar and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*,16:26-33,1996.

[14] N. Gura, A. Patel, A. Wander, H. Eberle and S. C. Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs . *Proceedings of Cryptographic Hardware and Embedded Systems - CHES'04*, LNCS 3156, pp. 119 - 132, Springer-Verlag, 2004

[15] M. E. Kaihara and N. Takagi. Bipartite modular multiplication. *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2005*, number 3659 in Lecture notes in Computer Science, pages 201-210, September 2005. Springer-Verlag.

[16] K. Iwamura, T. Matsumoto, and H. Imai. High-speed implementation methods for RSA scheme. In R. A. Rueppel, editor, *Advances in Cryptology: Proceedings of EUROCRYPT 92*, number 658 in Lecture Notes in Computer Science, pages 221-238. Springer-Verlag, 1992.

[17] L. Batina and G. Muurling. Montgomery in practice: How to do it more efficiently in hardware. In B. Preneel, editor, *Proceedings of RSA 2002 Cryptographers Track*, number 2271 in Lecture Notes in Computer Science, pages 40-52, San Jose, USA, February 18-22 2002. Springer-Verlag.

[18] T. Blum and C. Paar. Montgomery modular exponentiation on reconfigurable hardware. *In Proceedings of 14th IEEE Symposium on Computer Arithmetic*, pages 70-77, Adelaide, Australia, April 14-16 1999.

[19] S. H. Tang, K. S. Tsui and P. H. W. Leong. Modular exponentiation using parallel multipliers. *Proceedings of the 2003 IEEE International Conference on Field Programmable Technology (FPT)*, Tokyo, 52-59. 2003

[20] P. Schaumont and I. Verbauwhede. Interactive cosimulation with partial evaluation. *Proc. Design Automation and Test in Europe (DATE 2004),* pp. 642-647, 2004.

[21] H. Cohen, A. Miyaji and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. *Asiacrypt'98*, LNCS 1514, pp. 51-65, Springer-Verlag, 1998.

[22] M. Brown, D. Hankerson, J. López and A. Menezes. Software implementation of the NIST elliptic curves over prime fields. *Topics in Cryptology, CT-RSA 2001*, LNCS 2020, pp. 250-265, Springer-Verlag, 2001.